Page Faults

Page Faults — handling

Cause: access to page currently not present in main memory
$\longrightarrow$ exception, invoking OS

Process:
— OS checks access validity (requiring additional info)
— get empty frame
— load contents of requested page from disk into frame
— adapt page table
— set present bit of respective entry
— restart instruction causing page fault

Page Faults — latency

fault rate $0 \leq p \leq 1$
— $p = 0$: no page faults
— $p = 1$: every reference leads to page fault

effective access time (EAT):
EAT $= (1 - p) *$ memory access $+ p *$ (PF overhead $+$ PF service time $+$ restart overhead)

Page Faults — performance impact

memory access time: 200ns

average page fault service time: 8ms

$\rightsquigarrow$ 1:1000 access-page-fault-rate $\longrightarrow$ EAT $= 8.2\mu s \Rightarrow$ *slowdown by factor 40*!

Page Faults — challenges

what to eject?
— how to allocate frames among processes?
— which particular process's pages to keep in memory?
— see *page frame allocation*

what to fetch?
— what if block size $\neq$ page size?
— just one page needed? prefetch more?

process resumption?
— need to save state + resume
— process might have been in middle of instruction

Page Faults — what to fetch?

bring in page causing fault

pre-fetch sourrounding pages?
— reading two disk blocks is approximately as fast as reading one
— as long as there is no track/head switch, seek (disk) time dominates
— application exhibits spatial locality = big win

pre-zero pages?
— don't want to leak information between processes
— need 0-filled pages for stack, heap, .bss, …
— *zero on demand*?
— keep pool of 0-pages filled in background when CPU is idle?

Page Faults — process resumption?

hardware provides info about page fault
(intel: `%cr2` contains faulting virtual address)

context: OS needs to figure out fault context:
— read or write?
— instruction fetch?
— user access to kernel memory?

idempotent instructions: easy:
— re-do load/store instructions
— re-execute instructions accessing only one address

complex instructions: must be re-started
— some CISC instructions are hard to restart (e.g., block move of overlapping areas)
— *solutions*:
· touch relevant pages before operation starts
· keep modified data in registers $\longrightarrow$ page faults can't take place
· design ISA such that complex operations can execute partially $\longrightarrow$ consistent page fault state

Memory-Mapped Files — other issues

I/O mapping: mapping disk block to page in memory allows file I/O to be treated as routing memory
— *initial*: read page-sized portion of file from file system to physical page
— *subsequent read/write*: treated as ordinary memory access
$\longrightarrow$ *simplifies* file access, file I/O through memory instead of syscalls
$\longrightarrow$ *memory-file sharing*: several processes can map to same file

Shared Data Segments

implementation:
— temporary, asynchronous memory-mapped files
— shared pages (with allocated space on backing store)

copy on write (COW):
— allows both parent and child process to initially share same memory pages
— only modified pages are copied $\longrightarrow$ more efficient process creation

Page Frame Allocation — local vs. global

global: all frames considered for replacement
— does not consider page ownership
— one process cannot get another process's frame
— does not protect process from a process that hogs all memory

local: only frames of faulting process are considered for replacement
— isolates processes/users
— separately determine how many frames each process gets

Fixed Allocation — equal vs. proportional

equal: all processes get same amount of frames

proportional: allocate according to process size
$s_i :=$ size of process p_i, $S := \sum s_i$, $m :=$ total number of frames
$\Rightarrow a_i := \frac{s_i}{S} m$ allocation for p_i

Fixed Allocation — priority allocation

= proportional allocation scheme using priorities rather than size

on page fault of P_i:
— select one of its frames for replacement or
— select frame from process with lower priority

Memory Locality

problem: background storage much slower than memory
— paging extends memory size using background storage
— *goal*: run near memory speed, not near background storage speed

Pareto principle: applies to working sets of processes
— 10% of memory gets 90% of references
— *goal*: keep those 10% in memory, rest on disk
— *problem*: how to identify those 10%?

Thrashing

problem: system is busy swapping pages in and out
- each time one page is brought in, another page, whose contents will soon matter, is thrown out
- *effect*: low CPU utilization, processes wait for pages to be fetched from disk
- *consequence*: OS thinks that it needs higher degree of multiprogramming

reasons:
- *no temporal locality* of access pattern — process doesn't follow Pareto principle
- *too much multiprogramming*: each process fits individually, but too many for system
- *memory too small* to hold hot memory of a single process (the 10%)
- *bad page replacement policy*

Working-Set Model

Δ := working-set window (fixed number of page references; e.g., 10000 instructions)

WSS_i := working set of process P_i
- total number of pages referenced in most recent Δ (varies in time)
- $\Delta \begin{cases} \text{too small} & \Rightarrow \text{will not encompass entire locality} \\ \text{too large} & \Rightarrow \text{will encompass several localities} \\ = \infty & \Rightarrow \text{will encompass entire program} \end{cases}$

$D := \sum \text{WSS}_i$ = total demand for frames
- $D > m \rightsquigarrow$ **thrashing**

$\rightarrow D > m \Rightarrow$ suspend a process

Working Set — keeping track

perfect: replace page that is referenced furthest in the future (*oracle*)

idea: predict future from past
- record page references from past and extrapolate into future
- *problem*: too expensive to make ordered list of all page references at runtime

idea: sacrifice precision for speed
- MMU sets *reference bit* in respective page table entry every time a page is referenced
- set timer to scan all page table entries for reference bits

Page Fault Frequency — allocation scheme

goal: establish acceptable page fault rate
- *actual rate too low* $\longrightarrow$ give frames to other process
- *actual rate too high* $\longrightarrow$ allocate more frames to process

Page Fetch Policy — demand paging

idea: only transfer pages raising page faults

advantages:
- only transfer what is needed
- less memory needed by process $\longrightarrow$ higher multiprogramming degree possible

disadvantages:
- many initial page faults when task starts
- more I/O operations $\longrightarrow$ more I/O overhead

Page Fetch Policy — pre-paging

idea: speculatively transfer pages to RAM
- at every page fault: speculate what else should be loaded
- e.g., load entire text section when process starts

advantage: improves disk I/O throughput

disadvantages:
- wastes I/O bandwidth if page is never used
- can destroy working set of other processes in case of page stealing

Summary

paging simulates a memory size of the size of the virtual memory

when pages are filled via page faults, OS needs to answer some questions:
- what to eject?
- what to fetch?
- how to resume process?

different strategies to allocate frames and replace pages:
- local vs. global allocation
- fixed vs. proportional vs. priority allocation

thrashing must be prevented by taking working sets of active processes into account